Łamanie haseł metodą Brute-Force

Autorzy

Jacek Młynarczyk, 151747

Opis Projektu

Celem projektu było stworzenie programu do łamania haseł zaszyfrowanych SHA-256 metodą brute-force z wykorzystaniem przetwarzania równoległego. Projekt miał na celu przyspieszenie procesu łamania haseł poprzez zrównoleglenie operacji generowania i porównywania haseł, a także porównanie efektywności OpenMP i CUDA dla tego samego zadania.

OpenMP

Implementacja

Program korzysta z funkcji generującej hasła składające się z czterech znaków (małe i duże litery alfabetu oraz cyfry). Każde wygenerowane hasło jest następnie kodowane algorytmem SHA-256, a wynik jest porównywany z docelowym hashem.

```
#pragma omp parallel for private(current_password, current_hash) shared(found)
for (int i = 0; i < 62 * 62 * 62 * 62; i++) {
	if (found) continue;
	generate_password(i, current_password);
	sha256(current_password, current_hash);

	if (strcmp(current_hash, target_hash) == 0) {
		#pragma omp critical
		{
		if (!found) {
		found = 1;
		printf("Znaleziono hasło: %s\n", current_password);
			}
		}
	}
}
```

Pomiar czasu

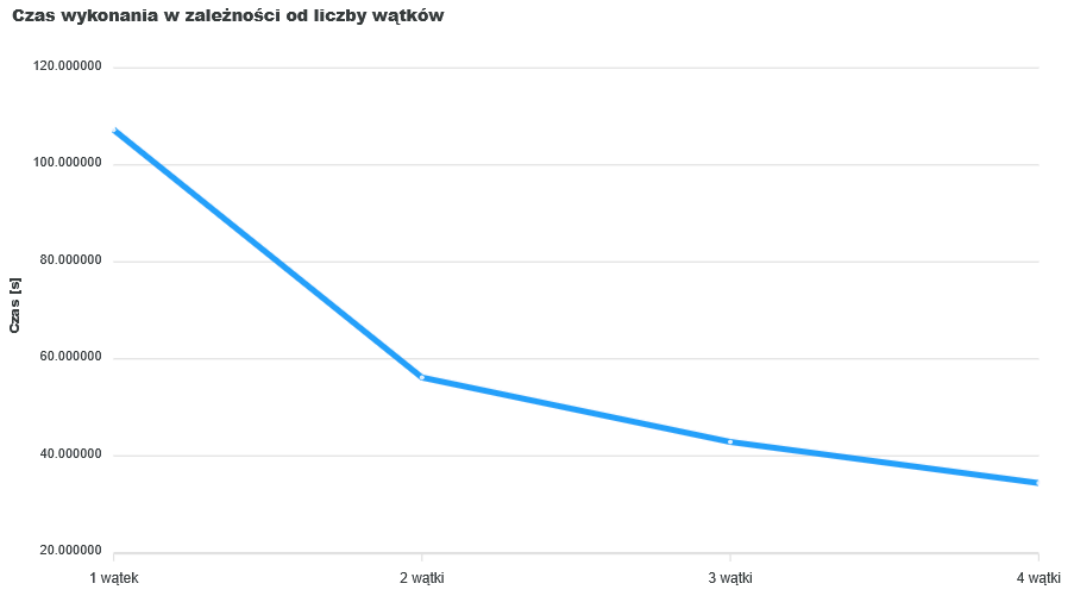
Do pomiaru czasu wykonania programu użyto funkcji omp_get_wtime(), która pozwala na precyzyjne zmierzenie czasu działania kodu zrównoleglonego:

```
double start_time = omp_get_wtime();
// Kod brute-force
double end_time = omp_get_wtime();
printf("Czas wykonania: %f sekund z %d wątkami\n", end_time - start_time, NUM_THREADS);
```

Testy i wyniki

Testy programu zostały wykonane dla różnych konfiguracji liczby wątków (1-4), aby zobaczyć, jak równoleglenie wpływa na czas wykonania. Poniżej znajdują się wyniki dla testów, w których próbowano złamać hasło „9999” (hash SHA-256: 888df25ae35772424a560c7152a1de794440e0ea5cfee62828333a456a506e05).

Liczba wątków	czas [s]
1	107,262491
2	56,150253
3	42,868202
4	34,373224



Na podstawie wyników można zauważyć, że wraz ze wzrostem liczby wątków czas wykonania skraca się niemal proporcjonalnie, co pokazuje, że problem łamania haseł brute-force jest dobrze skalowalny i łatwo poddaje się zrównolegleniu.

Analiza

Wykorzystanie OpenMP pozwoliło na znaczące przyspieszenie procesu łamania haseł metodą brute-force. Zwiększanie liczby wątków powodowało skrócenie czasu wykonania programu, a najlepsze wyniki osiągnięto przy wykorzystaniu maksymalnej liczby dostępnych wątków procesora.

CUDA

Sprzęt

Procesor: AMD Ryzen 7 7840HS, 3.5 GHz up to 5.1 GHz RAM: 32 GB LPDDR5, 6400MHz Karta graficzna: Nvidia RTX 4060

Implementacja

Główna funkcja brute_force_kernel rozdziela zakres obliczeń brute-force między tysiące wątków, co pozwala każdemu z nich generować różne hasła i porównywać je z docelowym hashem równolegle.

```
___global___ void brute_force_kernel(const char* target_hash, int* found, char* result) {

        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int total_threads = gridDim.x * blockDim.x;

        char current_password[PASSWORD_LENGTH + 1];
        char current_hash[65];

        for (int i = idx; i < CHARSET_SIZE * CHARSET_SIZE * CHARSET_SIZE * CHARSET_SIZE; i += total_threads) {

        if (*found) return; // Jeśli hasło zostało znalezione, zakończ wątek

        generate_password(i, current_password);
        sha256(current_password, current_hash);
        //printf("Thread %d: Index %d, Password: %s, Hash: %s\n", idx, i, current_password, current_hash);

        if (gpu_strcmp(current_hash, target_hash) == 0) {
        if (atomicExch(found, 1) == 0) {
        gpu_strcpy(result, current_password); // Zapisz znalezione hasło
        printf("Thread %d found the password: %s\n", idx, current_password);
        }
        return;
        }
        }

}
```

Zarządzanie pamięcią GPU

W CUDA pamięć GPU musi być ręcznie zarządzana przez programistę. Dlatego dane wejściowe, takie jak target_hash, są kopiowane z pamięci hosta do urządzenia przed rozpoczęciem obliczeń, a wyniki są przesyłane z powrotem po ich zakończeniu.

```
int* d_found;
	char* d_result;
	char h_result[PASSWORD_LENGTH + 1] = { 0 };
	int h_found = 0;

	char* d_target_hash;
	cudaMalloc((void**)&d_target_hash, (strlen(target_hash) + 1) * sizeof(char));
	cudaMemcpy(d_target_hash, target_hash, (strlen(target_hash) + 1) * sizeof(char),
cudaMemcpyHostToDevice);



	cudaMalloc((void**)&d_found, sizeof(int));
	cudaMalloc((void**)&d_result, (PASSWORD_LENGTH + 1) * sizeof(char));

	cudaMemcpy(d_found, &h_found, sizeof(int), cudaMemcpyHostToDevice);
[...]
	// Pobierz wynik
	cudaMemcpy(h_result, d_result, (PASSWORD_LENGTH + 1) * sizeof(char),
cudaMemcpyDeviceToHost);
	cudaMemcpy(&h_found, d_found, sizeof(int), cudaMemcpyDeviceToHost);

	// Zwolnij pamięć
	cudaFree(d_target_hash);
	cudaFree(d_found);
	cudaFree(d_result);
```

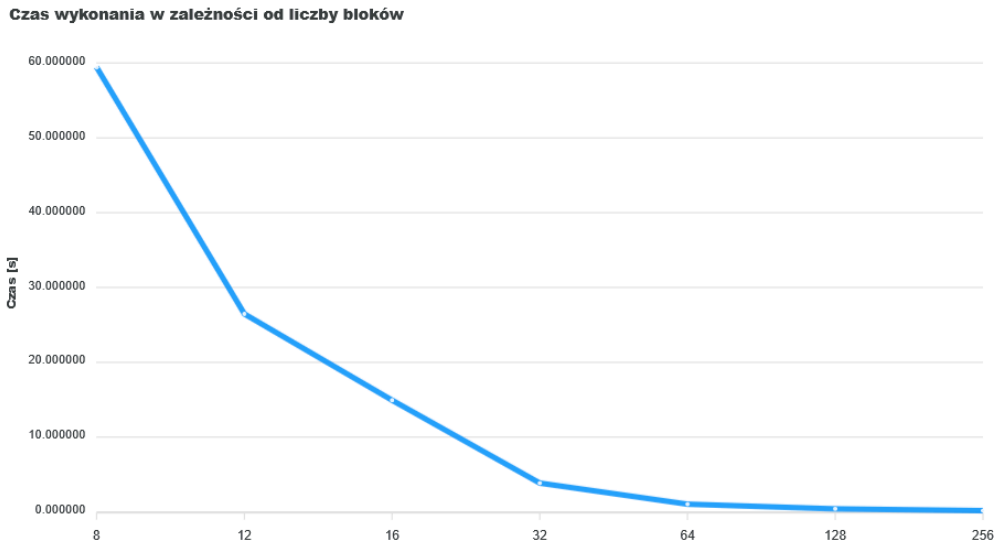
Pomiar czasu

```
	// Start liczenia czasu
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);
	cudaEventRecord(start);

	// Uruchom kernel
	brute_force_kernel << <blocks, threads_per_block >> > (d_target_hash, d_found, d_result);

	// Synchronizacja
	cudaDeviceSynchronize();

	// Koniec liczenia czasu
	cudaEventRecord(stop);
	cudaEventSynchronize(stop);
	float milliseconds = 0;
	cudaEventElapsedTime(&milliseconds, start, stop);
```

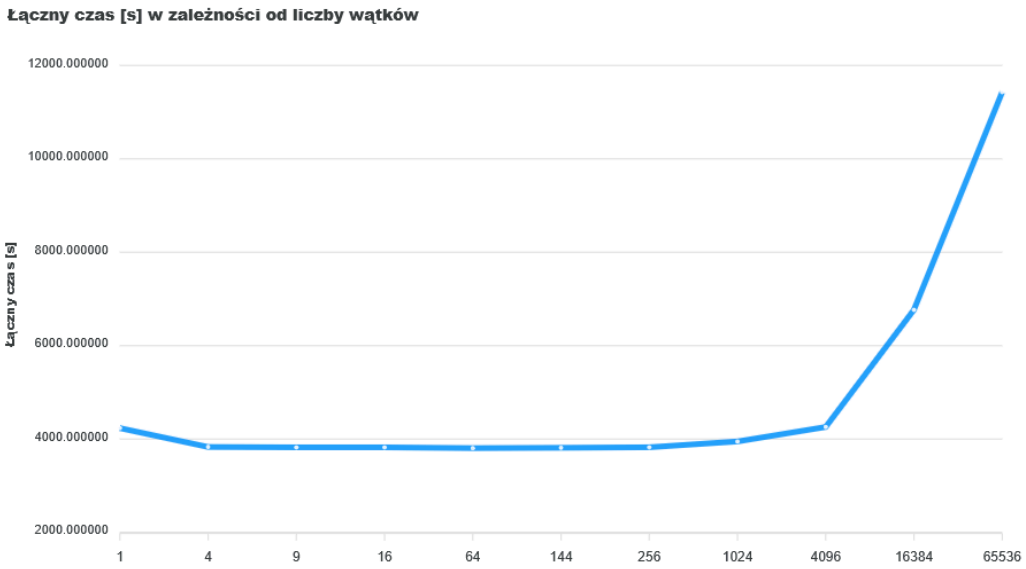
Testy i wyniki

Testy programu zostały wykonane dla różnych konfiguracji liczby bloków (1-256) oraz liczby wątków na pojedynczym bloku (1-256), aby móc porównać możliwości zrównoleglania łamania hasłą daną metodą. Poniżej znajdują się wyniki dla testów, w których próbowano złamać hasło „9999”. Dla uproszczenia i zwiększenia przejrzystości wykresów jako argument podawana jest liczba bloków.

Liczba bloków	Wątki na bloku	Wątki	czas [s]	Łączny czas [s]
1	1	1	4232,6055	4232,6055
2	2	4	957,360937	3829,443748
3	3	9	424,514875	3820,633875
4	4	16	238,848516	3821,576256
8	8	64	59,437184	3803,979776
12	12	144	26,471039	3811,829616
16	16	256	14,934396	3823,205376
32	32	1024	3,853822	3946,313728
64	64	4096	1,03939	4257,34144
128	128	16384	0,413046	6767,345664
256	256	65536	0,174434	11431,70662



Możemy również dokonać analizy, jak zmienia się łączny czas pracy wątków w zależności od ich liczby:



Widać, że mimo początkowego stałego czasu szukania hasła, to w momencie zbliżania się do całkowitego czasu poniżej 1 s, to zaczyna wzrastać łączny czas pracy. Jest to spodziewane ze względu na zwiększenie wkładu przetwarzania kodu, który nie jest ściśle związany z szukaniem hasła. Nie wpasowuje się w to jednak początek wykresu, gdzie dla 1 wątku łączny czas pracy również przewyższa czas dla większej liczby wątków. Może to być jednak związane z warunkami, w których program działał, gdyż pracował on ponad godzinę i funkcjonowanie sprzętu mogło go zakłócić.

Analiza

Jak pokazują wyniki, zwiększanie liczby wątków i bloków pozwala znacząco skrócić czas wykonania obliczeń brute-force dzięki efektywnemu wykorzystaniu równoległości GPU.

Wnioski

Oba podejścia bardzo się od siebie różnią, przede wszystkim w zakresie w jakim pozwalają pracować. OpenMP może efektywnie wykorzystywać tylko tyle wątków ile mamy rdzeni procesora, co jest znacząco ograniczające, jeżeli chcemy podzielić pracę na więcej wątków. CUDA pozwala wykorzystywać znacznie więcej wątków, co wielokrotnie przyspiesza przetwarzanie równoległe przy złożonych problemach. Widzimy jednak, że łączny czas pracy wszystkich wątków CUDA był znacząco dłuższy niż w OpenMP (Dla tego samego hasła CUDA osiągała 3800 s łącznej pracy, podczas gdy OpenMP tylko 107 s). Oczywiście nie jest to test przeprowadzony w odpowiednich warunkach, gdyż programy te korzystały z różnych kodów i pracowały na różnym sprzęcie. Ostatecznie CUDA okazało się wielokrotnie szybsze w wykonaniu obliczeń brute-force niż OpenMP, mimo że wymaga ręcznego zarządzania pamięcią. OpenMP może być bardziej wydajne dla mniejszych problemów, gdzie narzut związany z transferem danych między CPU a GPU jest dominujący.